OCTOBER 2014 VOLUME 29 NUMBER 10

Async Programming : Introduction to Async/Await on ASP.NET

Stephen Cleary (https://msdn.microsoft.com/en-us/magazine/mt149362?author=stephen+cleary) | October 2014

Most online resources around async/await assume you're developing client applications, but does async have a place on the server? The answer is most definitely "Yes." This article is a conceptual overview of asynchronous requests on ASP.NET, as well as a reference for the best online resources. I won't be covering the async or await syntax; I've already done that in an introductory blog post (bit.ly/19IkogW (http://bit.ly/19IkogW)) and in an article on async best practices (msdn.microsoft.com/magazine/jj991977 (https://msdn.microsoft.com/magazine/jj991977)). This article focuses specifically on how async works on ASP.NET.

For client applications, such as Windows Store, Windows desktop and Windows Phone apps, the primary benefit of async is responsiveness. These types of apps use async chiefly to keep the UI responsive. For server applications, the primary benefit of async is scalability. The key to the scalability of Node.js is its inherently asynchronous nature; Open Web Interface for .NET (OWIN) was designed from the ground up to be asynchronous; and ASP.NET can also be asynchronous. Async: It's not just for UI apps!

Synchronous vs. Asynchronous Request Handling

Before diving into asynchronous request handlers, I'd like to briefly review how synchronous request handlers work on ASP.NET. For this example, let's say the requests in the system depend on some external resource, like a database or Web API. When a request comes in, ASP.NET takes one of its thread pool threads and assigns it to that request. Because it's written synchronously, the request handler will call that external resource synchronously. This blocks the request thread until the call to the external resource returns. **Figure 1** illustrates a thread pool with two threads, one of which is blocked waiting for an external resource.

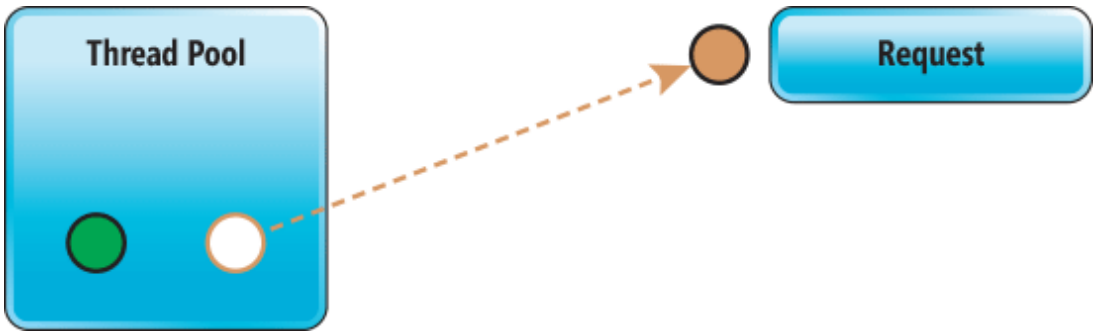

Figure 1 Waiting Synchronously for an External Resource

Eventually, that external resource call returns, and the request thread resumes processing that request. When the request is complete and the response is ready to be sent, the request thread is returned to the thread pool.

This is all well and good—until your ASP.NET server gets more requests than it has threads to handle. At this point, the extra requests have to wait for a thread to be available before they can run. **Figure 2** illustrates the same two-threaded server when it receives three requests.

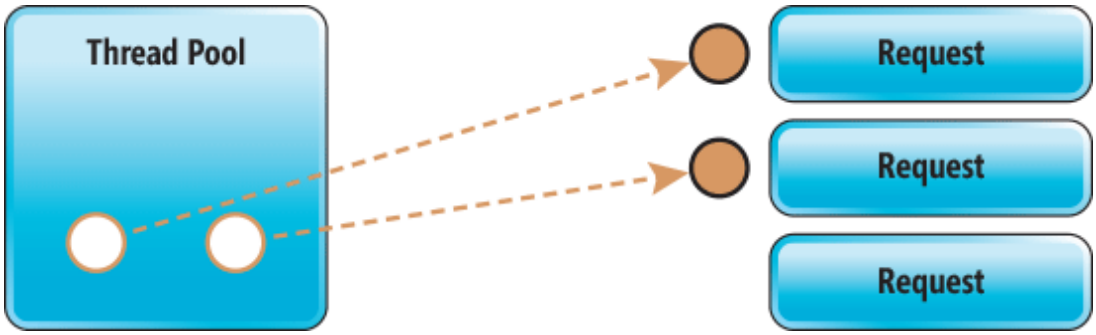

Figure 2 A Two-Threaded Server Receiving Three Requests

In this situation, the first two requests are assigned threads from the thread pool. Each of these requests calls an external resource, blocking their threads. The third request has to wait for an available thread before it can even start processing, but the request is already in the system. Its timer is going, and it's in danger of an HTTP Error 503 (Service unavailable).

But think about this for a second: That third request is waiting for a thread, when there are two other threads in the system effectively doing nothing. Those threads are just blocked waiting for an external call to return. They're not doing any real work; they're not in a running state and are not given any CPU time. Those threads are just being wasted while there's a request in need. This is the situation addressed by asynchronous requests.

Asynchronous request handlers operate differently. When a request comes in, ASP.NET takes one of its thread pool threads and assigns it to that request. This time the request handler will call that external resource asynchronously. This returns the request thread to the thread pool until the call to the external resource returns. **Figure 3** illustrates the thread pool with two threads while the request is asynchronously waiting for the external resource.

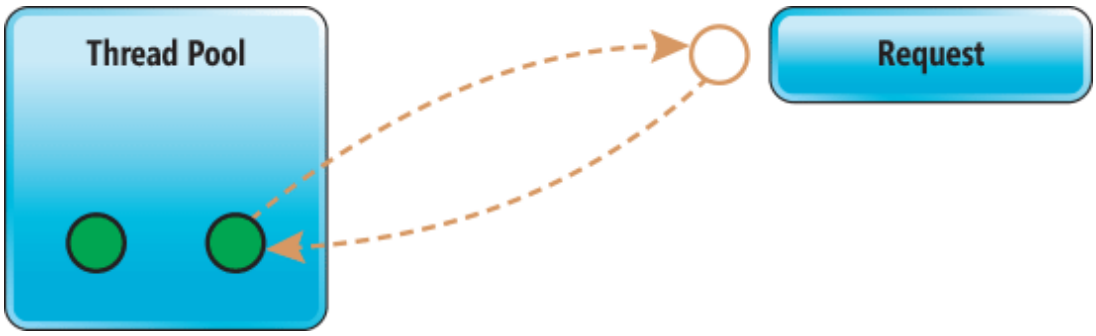

Figure 3 Waiting Asynchronously for an External Resource

The important difference is that the request thread has been returned to the thread pool while the asynchronous call is in progress. While the thread is in the thread pool, it's no longer associated with that request. This time, when the external resource call returns, ASP.NET takes one of its thread pool threads and reassigns it to that request. That thread continues processing the request. When the request is completed, that thread is again returned to the thread pool. Note that with synchronous handlers, the same thread is used for the lifetime of the request; with asynchronous handlers, in contrast, different threads may be assigned to the same request (at different times).

Now, if three requests were to come in, the server could cope easily. Because the threads are released to the thread pool whenever the request has asynchronous work it's waiting for, they're free to handle new requests, as well as existing ones. Asynchronous requests allow a smaller number of threads to handle a larger number of requests. Hence, the primary benefit of asynchronous code on ASP.NET is scalability.

Why Not Increase the Thread Pool Size?

At this point, a question is always asked: Why not just increase the size of the thread pool? The answer is twofold: Asynchronous code scales both further and faster than blocking thread pool threads.

Asynchronous code can scale further than blocking threads because it uses much less memory; every thread pool thread on a modern OS has a 1MB stack, plus an unpageable kernel stack. That doesn't sound like a lot until you start getting a whole lot of threads on your server. In contrast, the memory overhead for an asynchronous operation is much smaller. So, a request with an asynchronous operation has much less memory pressure than a request with a blocked thread. Asynchronous code allows you to use more of your memory for other things (caching, for example).

Asynchronous code can scale faster than blocking threads because the thread pool has a limited injection rate. As of this writing, the rate is one thread every two seconds. This injection rate limit is a good thing; it avoids constant thread construction and destruction. However, consider what happens when a sudden flood of requests comes in. Synchronous code can easily get bogged down as the requests use up all available threads and the remaining requests have to wait for the thread pool to inject new threads. On the other hand, asynchronous code doesn't need a limit like this; it's "always on," so to speak. Asynchronous code is more responsive to sudden swings in request volume.

Bear in mind that asynchronous code does not replace the thread pool. This isn't thread pool or asynchronous code; it's thread pool and asynchronous code. Asynchronous code allows your application to make optimum use of the thread pool. It takes the existing thread pool and turns it up to 11.

What About the Thread Doing the Asynchronous Work?

I get asked this question all the time. The implication is that there must be some thread somewhere that's blocking on the I/O call to the external resource. So, asynchronous code frees up the request thread, but only at the expense of another thread elsewhere in the system, right? No, not at all.

To understand why asynchronous requests scale, I'll trace a (simplified) example of an asynchronous I/O call. Let's say a request needs to write to a file. The request thread calls the asynchronous write method. WriteAsync is implemented by the Base Class Library (BCL), and uses completion ports for its asynchronous I/O. So, the WriteAsync call is passed down to the OS as an asynchronous file write. The OS then communicates with the driver stack, passing along the data to write in an I/O request packet (IRP).

This is where things get interesting: If a device driver can't handle an IRP immediately, it must handle it asynchronously. So, the driver tells the disk to start writing and returns a "pending" response to the OS. The OS passes that "pending" response to the BCL, and the BCL returns an incomplete task to the request-handling code. The request-handling code awaits the task, which returns an incomplete task from that method and so on. Finally, the request-handling code ends up returning an incomplete task to ASP.NET, and the request thread is freed to return to the thread pool.

Now, consider the current state of the system. There are various I/O structures that have been allocated (for example, the Task instances and the IRP), and they're all in a pending/incomplete state. However, there's no thread that is blocked waiting for that write operation to complete. Neither ASP.NET, nor the BCL, nor the OS, nor the device driver has a thread dedicated to the asynchronous work.

When the disk completes writing the data, it notifies its driver via an interrupt. The driver informs the OS that the IRP has completed, and the OS notifies the BCL via the completion port. A thread pool thread responds to that notification by completing the task that was returned from WriteAsync; this in turn resumes the asynchronous request code. There were a few threads "borrowed" for very short amounts of time during this completion-notification phase, but no thread was actually blocked while the write was in progress.

This example is drastically simplified, but it gets across the primary point: no thread is required for true asynchronous work. No CPU time is necessary to actually push the bytes out. There's also a secondary lesson to learn. Think about the device driver world, how a device driver must either handle an IRP immediately or asynchronously. Synchronous handling is not an option. At the device driver level, all non-trivial I/O is asynchronous. Many developers have a mental model that treats the "natural API" for I/O operations as synchronous, with the asynchronous API as a layer built on the natural, synchronous API. However, that's completely backward: in fact, the natural API is asynchronous; and it's the synchronous APIs that are implemented using asynchronous I/O!

Why Weren't There Asynchronous Handlers Already?

If asynchronous request handling is so wonderful, why wasn't it already available? Actually, asynchronous code is so good for scalability that the ASP.NET platform has supported asynchronous handlers and modules since the very beginnings of the Microsoft .NET Framework. Asynchronous Web pages were introduced in ASP.NET 2.0, and MVC got asynchronous controllers in ASP.NET MVC 2.

However, until recently, asynchronous code has always been awkward to write and difficult to maintain. Many companies decided it was easier all around to just develop the code synchronously and pay for larger server farms or more expensive hosting. Now, the tables have turned: in ASP.NET

4.5, asynchronous code using async and await is almost as easy as writing synchronous code. As large systems move into cloud hosting and demand more scale, more and more companies are embracing async and await on ASP.NET.

Asynchronous Code Is Not a Silver Bullet

As wonderful as asynchronous request handling is, it won't solve all your problems. There are a few common misunderstandings around what async and await can do on ASP.NET.

When some developers learn about async and await, they believe it's a way for the server code to "yield" to the client (for example, the browser). However, async and await on ASP.NET only "yield" to the ASP.NET runtime; the HTTP protocol remains unchanged, and you still have only one response per request. If you needed SignalR or AJAX or UpdatePanel before async/await, you'll still need SignalR or AJAX or UpdatePanel after async/await.

Asynchronous request handling with async and await can help your applications scale. However, this is scaling on a single server; you may still need to plan to scale out. If you do need a scale-out architecture, you'll still need to consider stateless, idempotent requests and reliable queueing. Async and await do help somewhat: they enable you to take full advantage of your server resources, so you won't have to scale out as often. But if you do need to scale out, you'll need a proper distributed architecture.

Async and await on ASP.NET are all about I/O. They really excel at reading and writing files, database records, and REST APIs. However, they're not good for CPU-bound tasks. You can kick off some background work by awaiting Task.Run, but there's no point in doing so. In fact, that will actually hurt your scalability by interfering with the ASP.NET thread pool heuristics. If you have CPU-bound work to do on ASP.NET, your best bet is to just execute it directly on the request thread. As a general rule, don't queue work to the thread pool on ASP.NET.

Finally, consider the scalability of your system as a whole. A decade ago, a common architecture was to have one ASP.NET Web server that talked to one SQL Server database back end. In that kind of simple architecture, usually the database server is the scalability bottleneck, not the Web server. Making your database calls asynchronous would probably not help; you could certainly use them to scale the Web server, but the database server will prevent the system as a whole from scaling.

Rick Anderson makes the case against asynchronous database calls in his excellent blog post, "Should My Database Calls Be Asynchronous?" (bit.ly/1rw66UB (http://bit.ly/1rw66UB)). There are two arguments that support this: first, asynchronous code is difficult (and therefore expensive in developer time compared to just purchasing larger servers); and second, scaling the Web server makes little sense if the database back end is the bottleneck. Both of those arguments made perfect sense when that post was written, but both arguments have weakened over time. First, asynchronous code is much easier to write with async and await. Second, the data back ends for Web sites are scaling as the world moves to cloud computing. Modern back ends such as Microsoft Azure SQL Database, NoSQL and other APIs can scale much further than a single SQL Server, pushing the bottleneck back to the Web server. In this scenario, async/await can bring a tremendous benefit by scaling ASP.NET.

Before Getting Started

The first thing you need to know is that async and await are only supported on ASP.NET 4.5. There's a NuGet package called Microsoft.Bcl.Async that enables async and await for the .NET Framework 4, but do not use it; it will not work correctly! The reason is that ASP.NET itself had to change the way

it manages its asynchronous request handling to work better with async and await; the NuGet package contains all the types the compiler needs but will not patch the ASP.NET runtime. There is no workaround; you need ASP.NET 4.5 or higher.

Next, be aware that ASP.NET 4.5 introduces a "quirks mode" on the server. If you create a new ASP.NET 4.5 project, you don't have to worry. However, if you upgrade an existing project to ASP.NET 4.5, the quirks are all turned on. I recommend you turn them all off by editing your web.config and setting httpRuntime.targetFramework to 4.5. If your application fails with this setting (and you don't want to take the time to fix it), you can at least get async/await working by adding an appSetting key of aspnet:UseTaskFriendlySynchronizationContext with value "true." The appSetting key is unnecessary if you have httpRuntime.targetFramework set to 4.5. The Web development team has a blog post on the details of this new "quirks mode" at bit.ly/1pbmnzK (http://bit.ly/1pbmnzK). Tip: If you're seeing odd behavior or exceptions, and your call stack includes LegacyAspNetSynchronizationContext, your application is running in this quirk mode. LegacyAspNetSynchronizationContext isn't compatible with async; you need the regular AspNetSynchronizationContext on ASP.NET 4.5.

In ASP.NET 4.5, all the ASP.NET settings have good default values for asynchronous requests, but there are a couple of other settings you might want to change. The first is an IIS setting: consider raising the IIS/HTTP.sys queue limit (Application Pools | Advanced Settings | Queue Length) from its default of 1,000 to 5,000. The other is a .NET runtime setting: ServicePointManager.DefaultConnectionLimit, which has a default value of 12 times the number of cores. The DefaultConnectionLimit limits the number of simultaneous outgoing connections to the same hostname.

A Word on Aborting Requests

When ASP.NET processes a request synchronously, it has a very simple mechanism for aborting a request (for example, if the request exceeded its timeout): It will abort the worker thread for that request. This makes sense in the synchronous world, where each request has the same worker thread from beginning to end. Aborting threads isn't wonderful for long-term stability of the AppDomain, so by default ASP.NET will regularly recycle your application to keep things clean.

With asynchronous requests, ASP.NET won't abort worker threads if it wants to abort a request. Instead, it will cancel the request using a CancellationToken. Asynchronous request handlers should accept and honor cancellation tokens. Most newer frameworks (including Web API, MVC and SignalR) will construct and pass you a CancellationToken directly; all you have to do is declare it as a parameter. You can also access ASP.NET tokens directly; for example, HttpRequest.TimedOutToken is a CancellationToken that cancels when the request times out.

As applications move into the cloud, aborting requests becomes more important. Cloud-based applications are more dependent on external services that may take arbitrary amounts of time. For example, one standard pattern is to retry external requests with exponential backoff; if your application depends on multiple services like this, it's a good idea to apply a timeout cap for your request processing as a whole.

Current State of Async Support

Many libraries have been updated for compatibility with async. Async support was added to Entity Framework (in the EntityFramework NuGet package) in version 6. You do have to be careful to avoid lazy loading when working asynchronously, though, because lazy loading is always performed synchronously. HttpClient (in the Microsoft.Net.Http NuGet package) is a modern HTTP client

designed with async in mind, ideal for calling external REST APIs; it's a modern replacement for HttpWebRequest and WebClient. The Microsoft Azure Storage Client Library (in the WindowsAzure.Storage NuGet package) added async support in version 2.1.

Newer frameworks such as Web API and SignalR have full support for async and await. Web API in particular has built its entire pipeline around async support: not only async controllers, but async filters and handlers, too. Web API and SignalR have a very natural async story: you can "just do it" and it "just works."

This brings us to a sadder story: Today, ASP.NET MVC only partially supports async and await. The basic support is there—async controller actions and cancellation work appropriately. The ASP.NET Web site has an absolutely excellent tutorial on how to use async controller actions in ASP.NET MVC (bit.ly/1m1LXTx (http://bit.ly/1m1LXTx)); it's the best resource for getting started with async on MVC. Unfortunately, ASP.NET MVC does not (currently) support async filters (bit.ly/1oAyHLc (http://bit.ly/1oAyHLc)) or async child actions (bit.ly/1px47RG (http://bit.ly/1px47RG)).

ASP.NET Web Forms is an older framework, but it also has adequate support for async and await. Again, the best resource for getting started is the tutorial on the ASP.NET Web site for async Web Forms (bit.ly/Ydho7W (http://bit.ly/Ydho7W)). With Web Forms, async support is opt-in. You have to first set Page.Async to true, then you can use PageAsyncTask to register async work with that page (alternatively, you can use async void event handlers). PageAsyncTask also supports cancellation.

If you have a custom HTTP handler or HTTP module, ASP.NET now supports asynchronous versions of those, as well. HTTP handlers are supported via HttpTaskAsyncHandler (bit.ly/1nWpWFj (http://bit.ly/1nWpWFj)) and HTTP modules are supported via EventHandlerTaskAsyncHelper (bit.ly/1m1Sn4O (http://bit.ly/1m1Sn4O)).

As of this writing, the ASP.NET team is working on a new project known as ASP.NET vNext. In vNext, the entire pipeline is asynchronous by default. Currently, the plan is to combine MVC and Web API into a single framework that has full support for async/await (including async filters and async view components). Other async-ready frameworks such as SignalR will find a natural home in vNext. Truly, the future is async.

Respect the Safety Nets

ASP.NET 4.5 introduced a couple of new "safety nets" that help you catch asynchronous problems in your application. These are on by default, and should stay on.

When a synchronous handler attempts to perform asynchronous work, you'll get an InvalidOperationException with the message, "An asynchronous operation cannot be started at this time." There are two primary causes for this exception. The first is when a Web Forms page has async event handlers, but neglected to set Page.Async to true. The second is when the synchronous code calls an async void method. This is yet another reason to avoid async void.

The other safety net is for asynchronous handlers: When an asynchronous handler completes the request, but ASP.NET detects asynchronous work that hasn't completed, you get an InvalidOperationException with the message, "An asynchronous module or handler completed while an asynchronous operation was still pending." This is usually due to asynchronous code calling an async void method, but it can also be caused by improper use of an Event-based Asynchronous Pattern (EAP) component (bit.ly/19VdUWu (http://bit.ly/19VdUWu)).

There's an option you can use to turn off both safety nets: HttpContext.AllowAsyncDuringSyncStages (it can also be set in web.config). A few pages on the Internet suggest setting this whenever you see these exceptions. I can't disagree more vehemently. Seriously, I don't know why this is even possible. Disabling the safety nets is a horrible idea. The only possible reason I can think of is if your code is already doing some extremely advanced asynchronous stuff (beyond anything I've ever attempted), and you are a multithreading genius. So, if you've read this entire article yawning and thinking, "Please, I'm no n00b," then I suppose you can consider disabling the safety nets. For the rest of us, this is an extremely dangerous option and should not be set unless you're fully aware of the ramifications.

Getting Started

Finally! Ready to get started taking advantage of async and await? I appreciate your patience.

First, review the "Asynchronous Code Is Not a Silver Bullet" section in this article to ensure async/await is beneficial to your architecture. Next, update your application to ASP.NET 4.5 and turn off quirks mode (it's not a bad idea to run it at this point just to make sure nothing breaks). At this point, you're ready to start true async/await work.

Start at the "leaves." Think about how your requests are processed and identify any I/O-based operations, especially anything network-based. Common examples are database queries and commands and calls to other Web services and APIs. Choose one to start with, and do a bit of research to find the best option for performing that operation using async/await. Many of the built-in BCL types are now async-ready in the .NET Framework 4.5; for example, SmtpClient has the SendMailAsync methods. Some types have async-ready replacements available; for example, HttpWebRequest and WebClient can be replaced with HttpClient. Upgrade your library versions, if necessary; for example, Entity Framework got async-compatible methods in EF6.

However, avoid "fake asynchrony" in libraries. Fake asynchrony is when a component has an async-ready API, but it's implemented by just wrapping the synchronous API within a thread pool thread. That is counterproductive to scalability on ASP.NET. One prominent example of fake asynchrony is Newtonsoft JSON.NET, an otherwise excellent library. It's best to not call the (fake) asynchronous versions for serializing JSON; just call the synchronous versions instead. A trickier example of fake asynchrony is the BCL file streams. When a file stream is opened, it must be explicitly opened for asynchronous access; otherwise, it will use fake asynchrony, synchronously blocking a thread pool thread on the file reads and writes.

Once you've chosen a "leaf," then start with a method in your code that calls into that API, and make it into an async method that calls the async-ready API via await. If the API you're calling supports CancellationToken, your method should take a CancellationToken and pass it along to the API method.

Whenever you mark a method async, you should change its return type: void becomes Task, and a non-void type T becomes Task<T>. You'll find that then all the callers of that method need to become async so they can await the task, and so on. Also, append Async to the name of your method, to follow the Task-based Asynchronous Pattern conventions (bit.ly/1uBKGKR (http://bit.ly/1uBKGKR)).

Allow the async/await pattern to grow up your call stack toward the "trunk." At the trunk, your code will interface with the ASP.NET framework (MVC, Web Forms, Web API). Read the appropriate tutorial in the "Current State of Async Support" section earlier in this article to integrate your async code with your framework.

Along the way, identify any thread-local state. Because asynchronous requests may change threads, thread-local state such as ThreadStaticAttribute, ThreadLocal<T>, thread data slots and CallContext.GetData/SetData will not work. Replace these with HttpContext.Items, if possible; or you can store immutable data in CallContext.LogicalGetData/LogicalSetData.

Here's a tip I've found useful: you can (temporarily) duplicate your code to create a vertical partition. With this technique, you don't change your synchronous methods to asynchronous; you copy the entire synchronous method and then change the copy to be asynchronous. You can then keep most of your application using the synchronous methods and just create a small vertical slice of asynchrony. This is great if you want to explore async as a proof-of-concept or do load testing on just part of the application to get a feeling for how your system could scale. You can have one request (or page) that's fully asynchronous while the rest of your application remains synchronous. Of course, you don't want to keep duplicates for every one of your methods; eventually, all the I/O-bound code will be async and the synchronous copies can be removed.

Wrapping Up

I hope this article has helped you get a conceptual grounding in asynchronous requests on ASP.NET. Using async and await, it's easier than ever to write Web applications, services and APIs that make maximum use of their server resources. Async is awesome!

Stephen Cleary *is a husband, father and programmer living in northern Michigan. He has worked with multithreading and asynchronous programming for 16 years and has used async support in the Microsoft .NET Framework since the first community technology preview. His homepage, including his blog, is at stephencleary.com (http://stephencleary.com/).*

Thanks to the following Microsoft technical expert for reviewing this article: James McCaffrey

MSDN Magazine Blog (http://blogs.msdn.com/msdnmagazine/)

More MSDN Magazine Blog entries > (http://blogs.msdn.com/msdnmagazine/)

Current Issue

(https://msdn.microsoft.com/en-us/magazine/dn846708) Browse All MSDN Magazines (https://msdn.microsoft.com/en-us/magazine/mt668395)

Subscribe to MSDN Flash newsletter (https://msdn.microsoft.com/en-us/aa570311.aspx?ocid=msdn_magazine)

Receive the MSDN Flash e-mail newsletter every other week, with news and information personalized to your interests and areas of focus.

Follow us Is this page helpful? Yes No Sign up for the MSDN Newsletter

Dev centers

Windows

Office

Visual Studio

Learning resources

Microsoft Virtual Academy (http://www.microsoftvirtualacademy.com/)

Channel 9 (http://channel9.msdn.com/)

Community

Forums (https://social.msdn.microsoft.com/forums/en-us/home)

Blogs (https://blogs.msdn.com/b/developer-

Support

Self support (https://msdn.microsoft.com/hh361695